rSYBL: a Framework for Specifying and Controlling Cloud Services Elasticity

Georgiana Copil, Daniel Moldovan, Hong-Linh Truong, Schahram Dustdar
, Distributed Systems Group, TU Wien, Austria

Cloud applications can benefit from the on-demand capacity of cloud infrastructures, which offer computing and data resources with diverse capabilities, pricing and quality models. However, state-of-the-art tools mainly enable the user to specify "if-then-else" policies concerning resource usage and size, resulting in a cumbersome specification process that lacks expressiveness for enabling the control of complex multi-level elasticity requirements.

In this paper, first we propose SYBL, a novel language for specifying elasticity requirements at multiple levels of abstraction. Second, we design and develop the rSYBL framework for controlling cloud services at multiple levels of abstractions. To enforce user-specified requirements, we develop a multi-level elasticity control mechanism enhanced with conflict resolution. rSYBL supports different cloud providers and is highly extensible, allowing service providers or developers to define their own connectors to the desired infrastructures or tools. We validate it through experiments with two distinct services, evaluating rSYBL over two distinct cloud infrastructures, and showing the importance of multi-level elasticity control.

CCS Concepts: •**Computer systems organization** → **Cloud computing;** •**Computing methodologies** → *Planning with abstraction and generalization;* Planning for deterministic actions; •**Software and its engineering** → Software design engineering;

Additional Key Words and Phrases: cloud computing, elasticity, elasticity requirements, control

1. INTRODUCTION

Web applications, workflows, and scientific applications can be offered as cloud services [Fard et al. 2012; Tsoumakos et al. 2013]. When deploying them (ideally, automatically) on various cloud infrastructures, the cloud service provider/developer usually has high level goals, e.g., testing reliability or achieving a specific level of performance with a minimum cost, at different levels of the service, e.g., for the entire data end or for specific parts of the data end. Current control frameworks mainly focus on single types of services and enable the provider/developer to only specify resource level SLA [Kouki et al. 2014]. Furthermore, they lack means to interact with the stakeholders (e.g., service provider, service developer) for controlling elasticity-related tradeoffs, e.g., the service provider cannot change requirements during runtime [Almeida et al. 2014].

As there are various types of stakeholders interested in cloud-hosted services (e.g., cloud service developers and cloud service providers), they might have different preferences at various abstraction levels. They have coarse or fine grained knowledge about parts of their services. For instance, the provider knows how much s/he is willing to pay for the entire service to be hosted on the cloud, while the developer knows quality

This work was partially supported by the European Commission in terms of the CELAR FP7 project (FP7-ICT-2011-8 #317790).
Author's addresses: TU Wien, Distributed Systems Group, Argentinierstrasse 8/184-1, A-1040 Vienna, Austria. Email: {e.copil, d.moldovan, truong, dustdar}@dsg.tuwien.ac.at.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. 1533-5399/2015/XXXX-ARTXXXX $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

indicators at different layers of the service. Therefore, there is a strong need for mechanisms to specify multi-level elasticity requirements, customized for various parts of the cloud service. To address these requirements, we need to develop means for multi-level elasticity requirements specification targeting high level goals referring to not only resources but, more importantly, to quality and cost, following the multi-dimensional definition of elasticity [Dustdar et al. 2011]. Moreover, we need to manage both the static description of the cloud service, and its runtime behavior, which depends on the virtual infrastructures on which it runs.

In this paper we present SYBL, a language for specifying elasticity requirements at different levels of abstraction in complex cloud services. We model various types of information for the elastic service, at runtime representing it as a relational graph which captures all the needed information for the cloud service control. We present our approach for multi-level elasticity control which generates action plans considering the evolution of the service at different levels of abstraction. To this end, we present our rSYBL framework, which is easily extensible, allows stakeholders to change their requirements during runtime, and supports multiple enforcement mechanisms (e.g., multiple clouds, and multiple software platforms), multiple monitoring tools, and planning mechanisms. We run experiments comparing rSYBL elasticity control on two cloud infrastructures, one private based on OpenStack[1], and the Flexiant[2] public cloud infrastructure. We showcase an experimental evaluation on the importance of multi-level service control, and analyze the performance of rSYBL under two different cloud infrastructures (i.e., OpenStack and Flexiant).

This paper substantially extends and details our previous work presented in [Copil et al. 2013b] and [Copil et al. 2013a] as follows: (i) we extend the service model from [Copil et al. 2013a] to support more detailed service description; (ii) we explain the information representation process, from creating the relational graph to its population with various types of information coming from multiple stakeholders; (iii) we detail our multi-level service control mechanisms; (iv) we describe the rSYBL framework; (v) we present three new experiments with rSYBL, showcasing its usefulness under multiple settings.

The paper structure is as follows: in Section 2 we present the cloud service model for describing different types of information related with the cloud service. In Section 3 we show the main characteristics of SYBL, Section 4 presents the algorithm that we use for generating action plans targeting multiple cloud service abstraction levels. Section 5 describes rSYBL framework and its main extensibility points, Section 6 presents experiments. Section 7 compares our work with existing research while Section 8 concludes the paper.

2. CLOUD SERVICE MODEL

2.1. Service units

Many types of scientific, enterprise and government cloud services have been emerging [Andrikopoulos et al. 2013; Inzinger et al. 2014], which mix a series of types of components, e.g., Machine-to-Machine (M2M) sensors, Web services/containers, and middleware. As shown in Figure 1, conceptually we can have a multitude of components running in the cloud, each with various capabilities. By using cloud technologies, on the one hand, each of these components can be re-configured during runtime. On the other hand, the cloud infrastructure also provides computing resources where they are executed and a series of capabilities for creating/modifying them. Therefore, these

[1] http://openstack.org/
[2] https://www.flexiant.com

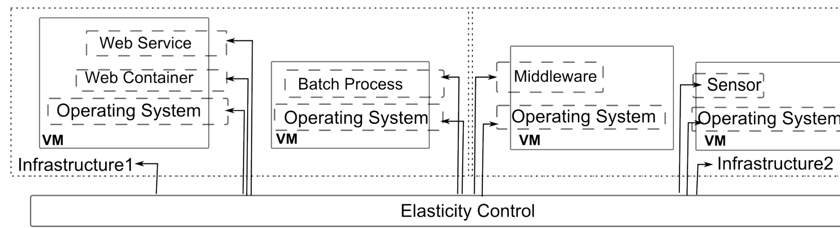

Fig. 1: Emerging cloud services control

components at runtime provide certain "service" capabilities, which we call "service units" [Tai et al. 2012].

Currently, most cloud control techniques scale only horizontally and at resource level the service unit (e.g., adding a new VM with the whole stack). However, understanding service units and their capabilities entails a highly granular control, using various types of control actions (e.g., change distribution mechanism for load balancing, change heap size, or change version), and combinations among them. These control actions can facilitate the fulfillment of a high range of requirements desired by cloud service stakeholders.

2.2. Elasticity requirements

Elasticity requirements are at the basis of cloud service elasticity, as they define the elasticity behavior that the cloud service stakeholder needs. Elasticity requirements are complex in the sense that they promote complex behavior for the elasticity of cost, quality and resources, through describing desirable states/behaviors in specific conditions. A complex cloud service can have multiple semantically connected service units, grouped into service topologies. Given this, elasticity requirements should refer to different cloud service parts (e.g., service unit, or service topology), and should be formulated at various granularities by various cloud service stakeholders. Current state of the art (see Section 7) facilitates description of low-level, infrastructure-related requirements. The cloud service stakeholder must be able to specify requirements concerning more abstract metrics (e.g., the cost per user that the stakeholder needs to pay per hour). We identify three types of elasticity requirements, which focus on the elasticity dimensions and the different dimensions among them: (i) cost-related elasticity requirements, (ii) quality-related elasticity requirements, and (iii) elasticity requirements on the relation between cost and quality. Cost and quality related requirements should specify expected values or expected policies under specific conditions. Requirements which focus on the relationship between cost and quality specify tradeoffs which are acceptable for stakeholders (e.g., a cloud service designer could need to specify that s/he is willing to pay more with 10% only if s/he receives a performance improvement of at least 20%)

Depending on the cloud service type, elasticity requirements might be associated with different parts of the cloud service, according to the cloud service structure. For the entire cloud service, one should specify requirements on aggregated metrics over the multiple parts of the cloud service (e.g., concerning the total cost). At service unit level one specifies requirements for that part of the service (e.g., a NoSQL data node), and all the services from the cloud provider it is using (e.g., all the virtual machines, or monitoring services). Moreover, stakeholders could specify requirements over service topologies, e.g., the reliability of the data topology of a multi-tier application should be very high, as it stores sensitive information. Considering services running continuously for a long time, these requirements might change due to various factors, such as business plans, popularity increase, or cloud provider cost updates.

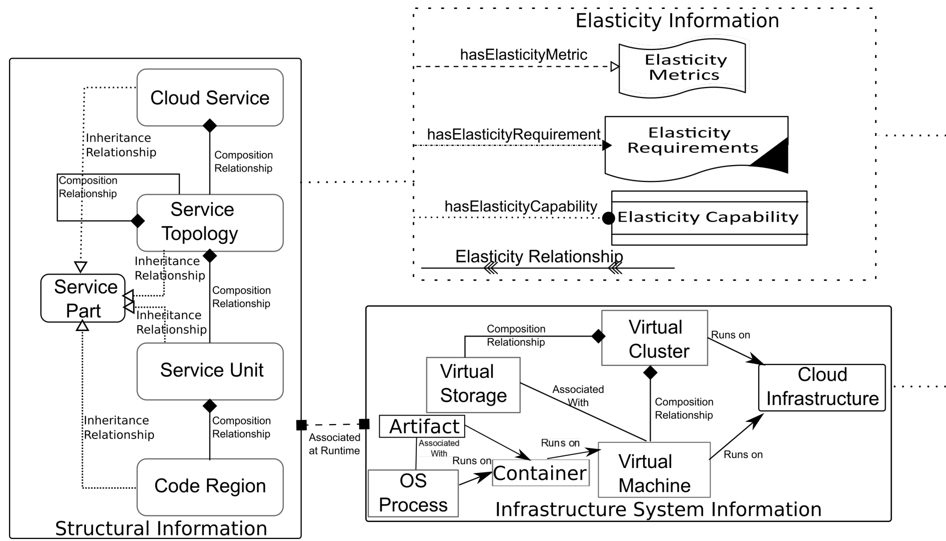

Fig. 2: Linking structural, elasticity and infrastructure system information

2.3. Cloud service structure

For specifying elasticity requirements at different abstraction levels, and then controlling elasticity at these levels, we need to know the structure and particularities of the cloud service. Current cloud service specification standards like TOSCA [3] and CIMI [4] facilitate the service description prior to the deployment, the description containing all the information needed for the deployment process. However, as the purpose of these languages is not to describe the cloud service runtime behavior, they cannot describe mechanisms to achieve elasticity at different levels. In order to generate and enforce control decisions during runtime, an elasticity controller would need to understand multiple types of information, e.g., information regarding cloud service units and the relation among them, information on the virtual resources used, or information regarding the cloud service developer/provider requirements. Therefore, we develop a representation model for our cloud service control, which overcomes the above-mentioned issues.

The cloud service description, shown in Figure 2, is designed to provide the cloud service elasticity controller with support for managing the cloud service. It holds different types of information: (i) structural/static information, (ii) virtual infrastructure related information, and (iii) elasticity related information. The cloud service can be seen as a graph composed of all this information, where each of the above concepts are nodes of the graph, descriptive information regarding the concept being modeled as node attributes and the relationships among them as edges connecting the various nodes.

The structural information describes the logical units out of which the cloud service is composed, and the relations between them:

— The *Cloud Service* represents the entire application or system, and can be further decomposed into service topologies and service units (e.g., a game, a web application, or a scientific application). The term cloud service that we choose to use is in accordance with existent cloud service architectures and standards (e.g.,TOSCA).

[3]https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca
[4]http://www.dmtf.org/sites/default/files/standards/documents/DSP0264_1.0.0.pdf

— The *Service Unit* [Tai et al. 2012] represents any kind of artifact, component or service offering computation and data capabilities (e.g., a web service, or a data analysis service).
— The *Service Topology* represents a logical grouping of service units that are semantically connected and that have elasticity capabilities as a group (e.g., a tier of a cloud service, or a part of a workflow).
— The *Code Region* represents a particular sequence of code for which the user can have elasticity requirements (e.g., a data analytics algorithm).

The infrastructure related information enables the elasticity controller to be aware which unit is deployed on which VM, or which cloud provider:
— *OS Processes* represent any kind of processes belonging to a cloud service that can be associated either with code regions or with service units (e.g., a web server process).
— *Artifact* is any atomic software unit or data set.
— *Containers* provide an additional layer of abstraction and automation (e.g., Docker[5], LXD[6])
— *Virtual Machine (VM)* and *Virtual Storage* are any IaaS services of type virtual machine and respectively storage which are purchased from the IaaS provider.
— The *Virtual Cluster* is a grouping of virtual machines or storage which have different properties (e.g., availability zone), and is offered as a service by the cloud provider.

This information regarding the infrastructure on which the cloud service is running is important in deciding how to control the service, since many of the actions depend on what the cloud provider offers. The above concepts (e.g., OS processes, or virtual cluster) are used to describe virtual resources in different cloud infrastructures[7]. This information regarding the infrastructure on which the cloud service is running is important in deciding how to control the service, since many of the actions depend on what the cloud provider offers.

The elasticity-related information facilitates the description of elasticity behavior for service units, service topologies or entire cloud service:
— *Elasticity Metrics* represent metrics targeted by elasticity requirements or lower-level metrics that are used for computing targeted metrics (e.g., cost vs. performance, cost vs. throughput, or cost vs. availability). Elasticity metrics can be associated with any cloud service part (e.g., service unit, service topology, or code region).
— *Elasticity Requirement*, represents any request coming from the user regarding elasticity of the cloud service (e.g., "the cost should not increase by more than 20% when the performance increases by less than 5%"). These requirements can be specified through SYBL and can be associated with any cloud service part.
— *Elasticity Capability*, represents any action/ mechanism/ operation through which the elasticity of the cloud service, of the service topology or of service units can be manipulated (e.g., the elastic reconfiguration of the data service topology for higher availability, or the elastic creation of new processing jobs for a map-reduce application).
— *Elasticity Relationship*, represents any connection between any two cloud service parts, which can be annotated with elasticity requirements (e.g., the connection between two service units needs to be of high reliability). We choose using the *re-*

[5]http://www.docker.com
[6]http://www.ubuntu.com/cloud/tools/lxd
[7]Even though some names might differ, the actual concept present high degree of similarity. E.g., Flexiant[2] uses `Server` for referring to VMs, offering `Disks` on the storage side, while Google Compute Engine[9] is offering various types of `Instances` (VMs), and `Storage`)

lationship term for being in accordance with cloud service specification standards (e.g., TOSCA).

We populate the graph constructed according to the model presented above with information from different sources (e.g., information from cloud providers regarding cloud infrastructure, pre-deployment information such as TOSCA description, or post-deployment associations between the static description and the virtual cloud infrastructure). Therefore, we do not assume that stakeholders will provide complete information at all the levels of the cloud service.

2.4. Managing Elasticity Capabilities from Cloud Providers

The model above uses the *Infrastructure System Information* for enabling the elasticity controller to describe, understand and manage the runtime information and its relation with service units, service topologies and the entire cloud service. All elements that are part of the Infrastructure System Information can have associated elasticity capabilities, described as part of the *Elasticity Information*. Most cloud providers implement similar concepts describing the services they offer (e.g., Flexiant[2] offers servers, Amazon[8] offers instances, and Google[9] offers virtual machine, Amazon offers Elastic Load Balancing while Google offers Global Load Balancing, although both refer to distributing incoming requests across pools of VMs). Therefore, we use the concepts presented in the above model for the Infrastructure System Information, in order to describe the services used from the chosen cloud providers. Moreover, we can abstract possible elasticity capabilities for all resources belonging to the Infrastructure System Information, in order to be referred by other elements of the model or by the cloud service elasticity controller.

All the elements which are part of Infrastructure System Information have associated, during runtime, (i) the properties which are used by the respective cloud service parts (e.g., service unit, service topology, or the entire cloud service), and from the Elasticity Information part, (ii) the elasticity capabilities which are available for the cloud service controller to manage them, together with the mechanisms for triggering these control capabilities, and (iii) elasticity metrics which give the controller the necessary information in order to take control decisions. The elasticity capabilities of the service units, service topologies and cloud services are composed of a list of elasticity capabilities of different resources from Infrastructure System Information and are enforced by the cloud service elasticity controller.

3. ELASTICITY REQUIREMENTS SPECIFICATION: SYBL

3.1. SYBL overview

The SYBL language for elasticity requirements specification is designed for specifying various types of requirements described in Section 2.2. SYBL facilitates the description of elasticity requirements at different levels, depending on the service provider's knowledge on the cloud service and on his/her perspective: cloud service, service topology, service unit, elasticity relationship, and programming/code region level. SYBL is implemented as directives in different languages, enabling easy description of the requirements, and delegating the actual difficult part of controlling the cloud service to the SYBL runtime (rSYBL), which is the controller of the cloud service.

Listing 1 shows in BNF the constructs of the SYBL language. The *monitoring* directives start with the `MONITORING` keyword and specify new variables to be monitored. A *constraint* defines elasticity requirements for the cloud service state, defining the limits of the cloud service behavior. A *strategy* specifies requirements on the elastic-

[8]http://aws.amazon.com/ec2
[9]https://cloud.google.com/compute/

ity behavior of the service. It specifies both control strategies to be enforced under specific conditions, and WAIT, STOP or RESUME actions for the controller, which can be paused/stopped/resumed when specified conditions hold. Therefore, with these two constructs at the center of the SYBL language, *constraints* and *strategies*, depending on the service provider/developer knowledge about the service, we enable various elasticity state and behavior specification mechanisms, the controllers interpreting the language, detailed in Section 4, being in charge with determining the specific control mechanisms which enable such service states or behaviors.

Listing 1: SYBL in Backus Naur Form (BNF)

```
Constraint := constraintName : CONSTRAINT ComplexCondition
Monitoring := monitoringName : MONITORING varName=MetricFormula
Strategy := strategyName : STRATEGY CASE ComplexCondition : action(
    parameterList)| strategyName : STRATEGY WAIT ComplexCondition|
        strategyName : STRATEGY STOP ComplexCondition|
        strategyName : STRATEGY RESUME ComplexCondition
MetricFormula := metric | number |varName| MetricFormula MathOperator metric
    | MetricFormula MathOperator number
ComplexCondition := Condition | ComplexCondition LogicalOperator Condition|(
    ComplexCondition LogicalOperator Condition)
Condition := metric RelationOperator number| number RelationOperator metric |
     Violated(name)|Fulfilled(name)
MathOperator := + | - | * | /
LogicalOperator := OR | AND | XOR | NOT
RelationOperator := <|>|>=|<=|==|!=
```

SYBL hides the complexity of enforcing a variety of complex calls, to different APIs (e.g., cloud provider APIs, or bash configurations) with the help of elasticity capabilities defined in the model in Section 2. It facilitates the service provider/developer to focus more on the elasticity requirements which would help his/her application to behave as desired. For referring to the current used infrastructure or platforms, it offers several predefined functions and environment variables with pre-defined semantics. The environment comprises different types of static and dynamic cloud information, its capabilities (e.g., whether or not it can modify the service during runtime and in what extent), as well as service-related information. When referring to the environment (e.g., through the predefined function $GetEnv$), the stakeholder needs to consider the level at which functions or variables appear, since information and extent of control varies with the level at which the SYBL elasticity requirement is specified. For instance, at service topology level the service provider/developer would get environment information regarding his/her service topology, which might be running in a different region than the rest of the cloud service.

3.2. Expressing SYBL requirements

The SYBL language is not strictly binded to a single implementation (e.g., requirements can be specified as Java annotations, C# annotations, or Python decorators). Moreover, the SYBL elasticity requirements can be injected into any cloud service description language (e.g., TOSCA) or can be specified separately through XML description. Current language interpretation mechanism is implemented in Java, and supports TOSCA-injected, XML-based, or Java annotation-based elasticity requirements specification.

For example, Listing 2 shows a constraint specified for the service topology with ID WebService Topology. The elasticity requirement sets the preferred response time below 450 ms. We define this elasticity requirement as a subtype of Java Annotation, triggered at runtime when the annotated method is executed and caught and interpreted using AspectJ.

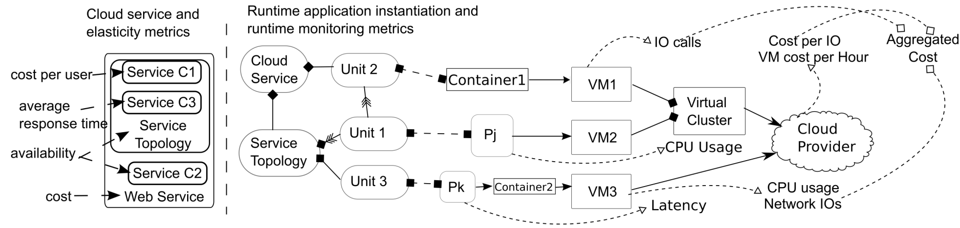

Fig. 3: Constructing runtime dependency graph

Listing 2: Example of elasticity requirements as Java Annotations

```
@SYBL_ServiceTopologyDirective(annotatedEntityID="WebServiceTopology",
    constraints="Co3:CONSTRAINT responseTime < 450 ms;")
```

Listing 3 shows a strategy for the service topology with ID `DataEnd Topology`. The elasticity requirement is a conditional strategy, which enforces the action `scalein` for the service topology when both responseTime and the average throughput are above predefined values.

Listing 3: Example of elasticity requirements in XML

```
<SYBLSpecification id="DataEndTopology" type="ServiceTopology">
    <Strategy Id="St1">
        <Condition>
            <BinaryRestriction Type="smallerThan">
                <LeftHandSide><Metric>throughputAverage</Metric></
                    LeftHandSide>
                <RightHandSide><Number>300</Number></RightHandSide>
            </BinaryRestriction>
            <BinaryRestriction Type="smallerThan">
                <LeftHandSide><Metric>responseTime</Metric></LeftHandSide>
                <RightHandSide><Number>360</Number></RightHandSide>
            </BinaryRestriction>
        </Condition>
        <ToEnforce ActionName="scalein" />
    </Strategy>
</SYBLSpecification>
```

The constraint shown in Listing 4 specifies that the cost for the `PilotCloudService` should be below 100$. The SYBL elasticity requirements can be easily integrated within TOSCA policies, and interpreted by the elasticity controller.

Listing 4: Example of elasticity requirements as TOSCA Policies

```
<tosca:ServiceTemplate name="PilotCloudService">
  <tosca:Policy name="St1" policyType="SYBLStrategy">
    St1:STRATEGY minimize(Cost) WHEN high(overallQuality)
  </tosca:Policy>...
```

4. MULTI-LEVEL ELASTICITY CONTROL

4.1. Runtime dependency graph of elastic cloud services

In order to describe the cloud service during runtime, a runtime elasticity dependency graph is used, which has as nodes the concepts described in the model presented in Section 2. This dynamic graph captures all the information about the structure, and runtime information like elasticity metrics, requirements and deployment topology during runtime and is constructed by our elasticity controller described in detail in the following sections. Initially, the elasticity dependency graph is populated with different types of information, in order to construct the knowledge base for elasticity

control. Moreover, the dependency graph is populated continuously with monitoring information, coming from MELA [Moldovan et al. 2013], Ganglia[10], or other monitoring tools. Figure 3 shows how the runtime dependency graph is constructed. If we take the example of a Web service (the left side of Figure 3), the cloud user views his/her Web service as a set of services (in this case `Service C1`, `Service C2`, and `Service C3`), some of them grouped together for monitoring purposes (in this case `Service Group` which consists of `Service C1` and `Service C3`). The metrics targeted in user's elasticity requirements in this stage are high level metrics, referring to the quality, cost and resources of services, of groups of services or even of the entire Web service. The right part of Figure 3 shows the dependency graph being constructed at runtime by our elasticity control. Service instances are deployed on virtual machines, in different virtual clusters and virtual providers, aggregating low-level metrics for computing higher level ones. For instance, the availability at service level would be computed from the availability at each service part and the cost is determined from cost per I/O and VM cost and the run-time service topology and loads.

4.2. Steps in multi-level elasticity control

Considering the model of the cloud service described through the runtime dependency graph presented in the Section 2, we enable elasticity control simultaneously for each of the described nodes, resulting in a multi-level elasticity control of the described cloud service. The service provider/developer describes his/her cloud service using TOSCA or other description standards. The initial deployment configuration is specified either by the automatic deployment tool used or by the service provider/developer if a manual deployment approach is chosen. The elasticity requirements are evaluated and conflicts which may appear among them are resolved. After that, an action plan is generated, consisting of elasticity capabilities which enable the fulfillment of specified elasticity requirements. The action plan is composed from elasticity capabilities that have associated a series of IaaS calls, configurations, or bash/scripts executions.

Let us consider a simple example shown in Figure 4 of controlling the entire cloud service, e.g., by the system designer. The described elasticity requirements, `Co1`, `Co2`, and `Co3` are not conflicting, and elasticity capabilities are searched for fulfilling these requirements. Possible elasticity capabilities are, for instance, for the case the running time is higher than 10 hours and the cost is still in acceptable limits to scale-out for the computation service topology, increasing the processing speed. An example of an action plan, shown in Figure 4 could be `ActionPlan1=[ [increaseReplication], [scaleOut, setThreadPool=100]]`. This action plan would address performance issues for the second elasticity requirement `Co2`, and availability issues for the third elasticity requirement `Co3`. Each of the generated elasticity capabilities are mapped into complex API calls. For instance, `increaseReplication` elasticity capability would consist of calls for adding and configuring a new database node and configuring the cluster for higher replication, while the `scaleOut` elasticity capability would be the addition of a new virtual machine, deployment of the `ComputationEnd` on the new machine, and necessary calls for the new instance of the service unit to join the computation topology cluster.

4.3. Resolving elasticity requirements conflicts and generating action plans

We identify two types of conflicts: (i) conflicts between elasticity requirements targeting the same abstraction level, and (ii) conflicts which appear between elasticity requirements targeting different abstraction levels. For the first case, sets of conflicting

[10] http://ganglia.sourceforge.net/

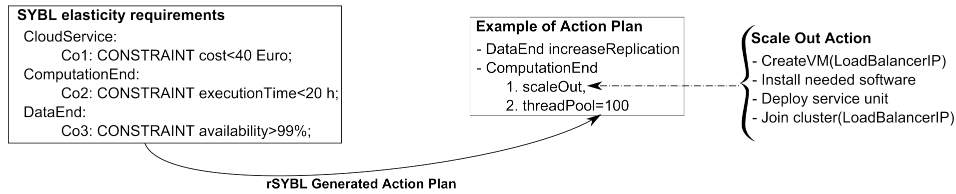

Fig. 4: An action plan example

constraints are identified and a new constraint overriding previous set is added to the dependency graph for each level. In the second type of conflict the constraints from a lower level (e.g., service unit level) are translated into the higher constraint's level (e.g., service topology level), by aggregating metrics considering the dependency graph. Since the problem is reduced to same-level conflicting elasticity requirements, we use the approach for the same-level conflicting elasticity requirements and compute a new elasticity requirement from overlapping conditions. More details on requirements resolution are available in [Copil et al. 2013a].

For generating an action plan for cloud service elasticity control, we formulate the planning problem as a maximum coverage problem: we need the minimum set of capabilities which help fulfilling the maximum set of requirements. Given the current cloud service state, we can apply a number of elasticity capabilities from the *Elasticity Capability Set* ECS. As described in Equation 1, for each elasticity capability enforcement, we reach a state with a set of requirements fulfilled EC_x. We therefore need the minimum set of capabilities which fulfill the maximum set of requirements. Since maximum coverage problem is an NP-hard problem, and our research does not target finding the optimal solution for it, we choose the greedy approach which offers an $1 - \frac{1}{e}$ approximation.

$$ECS = \{EC_1, EC_2, ..., EC_n\}$$
$$EC_x = \{Fulfilled(R_{x_1}), ..., Fulfilled(R_{x_y}) | R_i \in Requirements\} \tag{1}$$

The main step of the greedy approach consists of finding each time the elasticity capability EC_i fulfilling the most constraints and improving the most strategies. After selecting an elasticity capability in this iterative process, the ECS needs to be recomputed since the context of the service is changed and the effect of applying EC_j will be different than before applying EC_i. For now we consider that the effects of enforcing an elasticity capability are introduced by the user, our framework presented in Section 5 offering mechanisms for easily plugging-in tools that automatically detect the effect of an elasticity capability.

4.4. Enforcing action plans

For controlling the elasticity of cloud services, tools monitoring the elasticity and the different types of metrics targeted by the cloud service user are necessary. Although at the moment existing cloud APIs offer only access to low-level resources, elasticity control of cloud services would also impose the existence of cloud APIs which take into account the different levels of metrics or the cloud service structure.

For overcoming this situation, we use the MELA framework which aggregates low-level metrics for achieving higher level ones, and use existent resource-level control capabilities for manipulating higher level quality and cost. For instance, the cost of a service unit would be composed of the different types of cost associated to each resource associated with the service unit, like cost depending on the number of virtual machines, cost for intra-unit communication, or I/O cost. The cloud service cost is computed as the sum of service unit cost, inter-unit communication cost, and possible licensing costs.

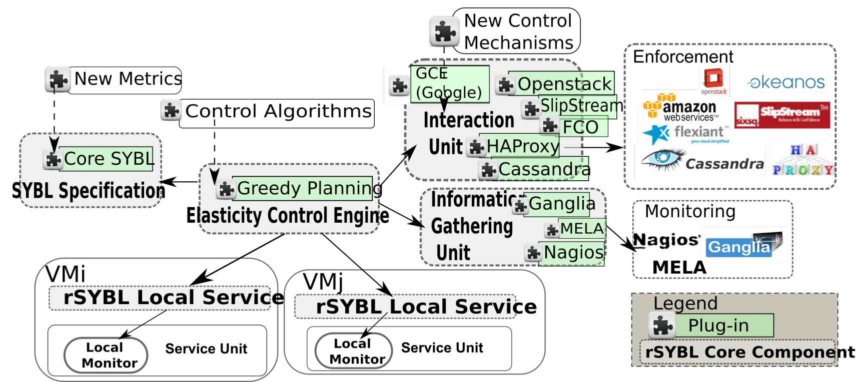

Fig. 5: Architecture of rSYBL

Considering long running services, the stakeholders can evaluate the actions generated, and revise their requirements on the basis of application behavior. This is possible either before or after action plan enforcement, rSYBL re-running the elasticity control loop, starting from the first step described in Section 4.1.

A roll-back mechanism for each capability allows the controller to also handle situations where the action plans do not produce the expected results when enforced. When this is observed, the reverse actions associated with each capability in the reverse order of the action plan enforcement.

5. RSYBL: ELASTICITY CONTROL AS A SERVICE

Based on the above-presented concepts and mechanisms, we develop the rSYBL framework[11], shown in the Figure 5. The central module of rSYBL is the *Control Service*, which takes processed elasticity requirements from *SYBL Specification* unit, and communicates with *Interaction Unit* for enforcing found elasticity mechanisms, and with *Monitoring Information Gathering Unit* for pulling monitoring and analysis information on the cloud service. The distributed components of rSYBL are the *Local Monitor* and the *Local Service*. The Local Monitor is part of the service units (e.g., in this case as a weaving library) and knows when a process with SYBL programming directives has started, or when a sequence of code annotated with SYBL has started or finished. The Local Monitor communicates with the Local Service for sending SYBL elasticity requirements, for ensuring elasticity requirements through local elasticity mechanisms, e.g., reconfigure the service unit to accommodate higher number of customers, or increase maximum thread pool size. The current rSYBL prototype supports elasticity control of cloud services, service units, service topologies, relationships and code regions for fulfilling elasticity requirements which can be specified in XML or through Java Annotations detected at runtime with AspectJ. We tested our prototype on our local cloud running OpenStack[1] using JClouds[12] for controlling virtual machines, and on Flexiant [2] cloud using the Flexiant Cloud Orchestrator (FCO) REST API, and using MELA for monitoring.

The rSYBL framework is designed to be easily extended, customized and used for various applications, in different environments and focusing on various metrics. We designed plugin mechanisms for different parts of the framework, as shown in Figure 5, currently being available various plugins for information gathering (e.g., Ganglia, and MELA), for interacting with cloud infrastructures and service artifacts (e.g.,

[11]Find the prototype implementation and further experiments at http://dsg.tuwien.ac.at/research/viecom/SYBL
[12]http://jclouds.org/

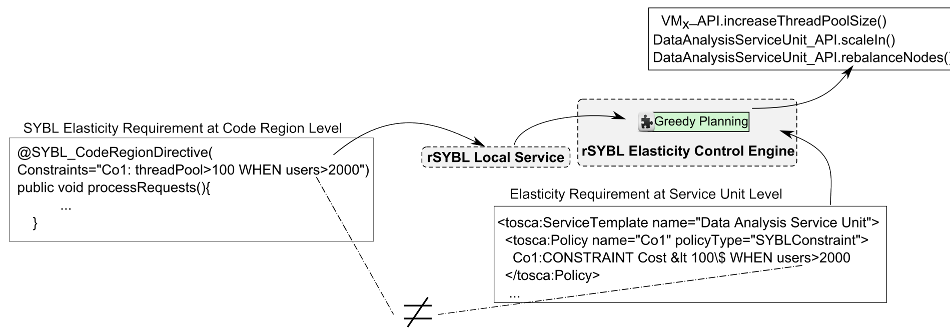

Fig. 6: Elasticity requirements processing by rSYBL

OpenStack, FCO, or Cassandra), and for control algorithms (e.g., the greedy planning described in Section 4.3). Firstly, the *specification* of SYBL directives can be extended from the point of view of the metrics that we use, and of the higher level metrics that are defined. The link of the new metrics names with the manner of finding them in the new plugins is done through a simple configuration file. Both the *monitoring and analysis* and the *enforcement* units can be adapted for working in different environments. For instance, in one extreme, one may want to control the elasticity of a service which is deployed on one or several local servers. This may be the case of a service provider who is interested in deploying just parts of its service on the cloud, but on elastically controlling all the cloud service. In this case, the service units which are deployed on the cloud benefit from more mechanisms of elasticity control than the local ones, but all can be elastically controlled with rSYBL by just specifying the control mechanisms for each service unit or service topology which can be controlled. On the other hand, even if the entire service is deployed on the cloud, we may need monitoring information from different sources, e.g., one API may provide process-level information, other API may provide quality or cost related information, and other one could provide VM-level monitoring information. All this information can be used at the same time by rSYBL, by using plugins for each mechanism of information gathering. These plugins will be used by rSYBL in evaluating elasticity requirements, learning about the cloud service behavior, and planning for next control mechanisms to be used.

The Control Service is deployed on per cloud infrastructure basis, and can control multiple cloud services at once. For catching the events sent by the Local Monitor which handles programming level SYBL elasticity requirements, we need to also deploy on each virtual machine a Local Service instance which is part of rSYBL, and to use inside our controlled application the Local Monitor library. The rSYBL Control Service, Interaction Unit and Information Gathering Unit are components of rSYBL core, being deployed on the same virtual machine and connecting to the necessary tools for monitoring and enforcement, depending on the available plug-ins. Figure 5 shows the current plug-ins used by rSYBL for elasticity control.

5.1. Linking elasticity requirements to enforced actions

Figure 6 shows a flow from elasticity requirements specification to the enforcement of actions, which are mapped from an elasticity capability of scaling up at thread level and one of scaling in at data analysis service unit level. The Java annotation based elasticity requirement is injected by the cloud service developer into code, from where an rSYBL library (the `Local Monitor`) weaves it when the annotation is triggered, and forwards it to the rSYBL `Local Service` which is deployed on each VM for providing local control. The `Local Service` processes the triggered requirement, checks whether or not it can enforce it locally, and if it cannot be enforced locally sends it forward to the

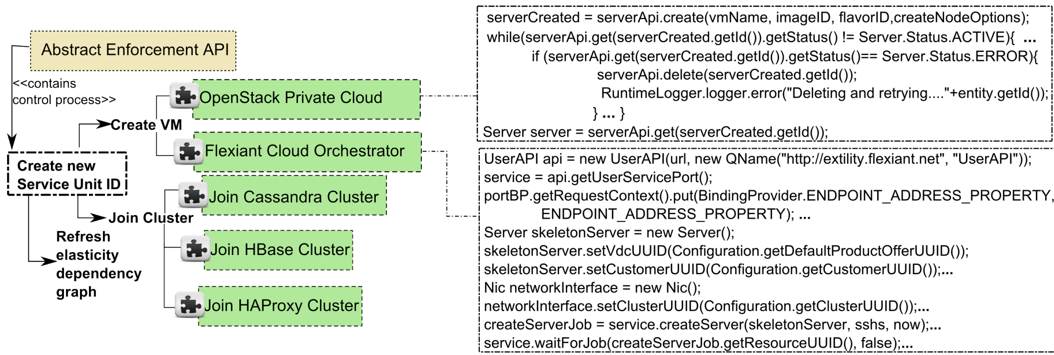

Fig. 7: Provided elasticity control plugins in rSYBL

Level	Elasticity Capability	Action
Infrastructure	Scale In	Remove Network Interface; Remove VM
	Scale Out	Create Network Interface; Create Disk; Create VM
	Custom Action	Attach/detach disk, scale vertically
Platform	Scale In	Leave Cluster; Remove Artifact
	Scale Out	Create Artifact; Join Cluster
	Reconfiguration	Increase Thread Pool; Decrease Thread Pool;Set Load Distribution Mechanism
Application	Reconfiguration	Set Specific Config. Param.
	Custom Action	Action Given by User

Table I: Example of elasticity capabilities at different control levels

`Control Service`. The `Control Service` also receives the SYBL elasticity requirement which is described as TOSCA policy. It evaluates the received elasticity requirements, and in case it finds suitable elasticity capabilities which are expected better fulfill de requirements, it triggers their enforcement. As described in Section 4, the enforcement also consists of the mapping of each of these elasticity capabilities to cloud-provided or cloud service specific APIs, which we detail in what follows.

5.2. Elasticity capabilities used in the elasticity control process

rSYBL facilitates the control through various types of elasticity capabilities, described through the model presented in Section 2, which are either exposed by IaaS providers, or by software used by the cloud service. Action plans are saved in a repository, and reused for learning their effects on the service behavior. Figure 7 shows how rSYBL facilitates the description of different complex actions which involve complex calls (right-hand side of Figure 7) for a variety of infrastructures and platforms, in order for the rSYBL common user to specify simple elasticity capabilities which hide a lot of complexity. Moreover, the rSYBL user (e.g., a cloud service developer) can specify custom actions (e.g., through TOSCA plans) to be executed by the controller as standalone or as part of an elasticity capability.

In the SYBL strategies, the user can simply specify the elasticity capability name, while the rSYBL controller is in charge with detecting the exact combination of actions necessary, from the ones presented in Table I). The table is not meant to be exhaustive, and new actions are continuously developed by cloud providers. At infrastructure level for example, recently some of the providers have allowed attaching/detaching disks without the need to restart the corresponding virtual machines

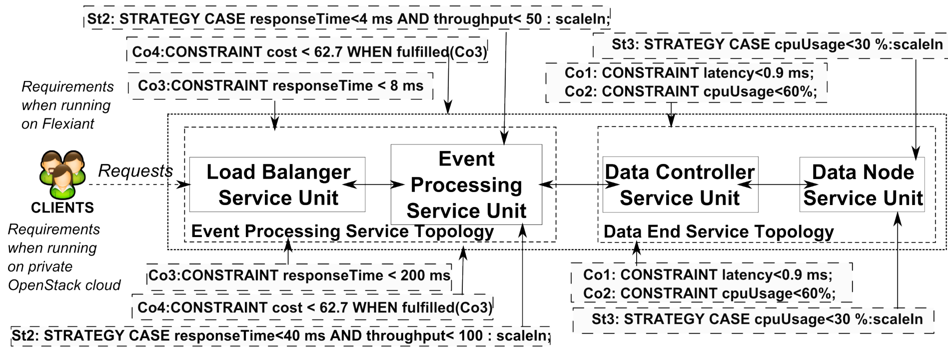

Fig. 8: M2M DaaS with SYBL elasticity requirements

(e.g., Flexiant). For enforcing platform-level scale in, we have two options: either focusing on platform level actions (e.g., `leave cluster` & `remove artifact`, `remove network interface`), or also using infrastructure-level actions (e.g., `leave cluster`, `remove VM`, `remove network interface`). rSYBL will decide on the appropriate action, depending on the expected effect and whether or not it is possible to collocate more artifacts on the same VM without the need for IaaS level actions. The supported platform software can be extended by simply implementing the described actions in plugins. For the case of the application-level actions, the user can decide to customize rSYBL by implementing plugins or to call deployment-defined actions from SYBL strategies.

For implementing custom plugins, or supporting new monitoring/enforcement plugins, three steps are necessary: (i) implementing the monitoring/enforcement interfaces from rSYBL, (ii) adding the needed configurations (i.e., credentials or other plugin-specific information), and (iii) adding the primitive actions offered by plugins to the primitive actions description. The SYBL directives are not dependent on the plugins available, as capabilities are composed of primitives associated with the service or plugins. With this indirection layer, there is no need for changes in the SYBL directives when plugins are added or removed to rSYBL.

More technical details on configuring and running the framework are available on the rSYBL wiki[13].

6. EXPERIMENTS

6.1. Controlling elasticity with rSYBL: M2M DaaS Cloud Service

Considering a machine-to-machine (M2M) DaaS[14] which processes information coming from various data sensors, we design the application in Figure 8, and simulate clients which send sensor information. Specifically, the M2M DaaS is comprised of an *Event Processing Service Topology* and a *Data End Service Topology*. The *Event Processing Service Topology* consists of a *Load Balancer Service Unit* and an *Event Processing Service Unit*, which analyzes and stores data in a NoSQL cluster, in this case Cassandra-based, composed of a *Data Controller Service Unit* (i.e., Cassandra seed) and a *Data Node Service Unit* (i.e., Cassandra node).

We deploy the M2M DaaS service on two different cloud infrastructures: (i) using Flexiant[2] cloud provider, we deploy on their public cloud, and (ii) on our OpenStack[1]-based private cloud. We simulate data sensors which send information to *Event Processing Service Topology* with a python-based load generator which sends random data to our M2M DaaS. As the two clouds considered are different, they differ in relia-

[13]https://github.com/tuwiendsg/rSYBL/wiki
[14]M2M DaaS prototype: https://github.com/tuwiendsg/DaaSM2M

Setting	Flexiant	OpenStack
Small instance GB:CPU	2GB:2CPU	1GB:1CPU
Small instance price	6 Flexiant Units/h	3 OpenStack Units/h
Network interface card price	0.13 Flexiant Units/h	0.13 OpenStack Units/h

Table II: Experiment Unit Costs

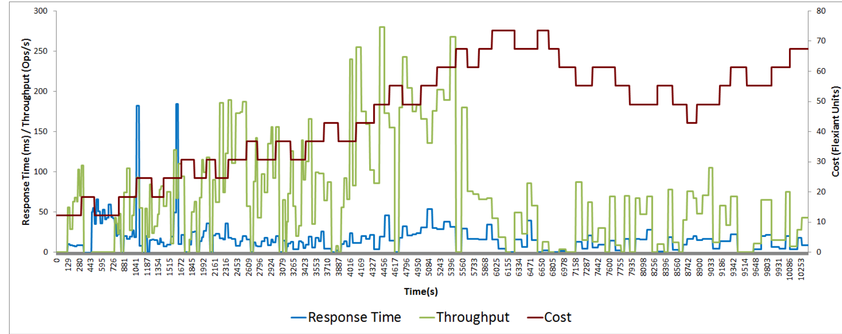

Fig. 9: Event Processing Service Topology on Flexiant public cloud

bility, in the estimated cost, and in quality characteristics, even when using similar resources. Table II shows the settings of our experiments in terms of estimated cost. The Flexiant cloud provider costs vary with the user type, and it is manually set by Flexiant cloud administrators. For our case, the price for an instance with 2 GB and 2 CPU is 6 units per hour, where the units can be bought at varying prices (from 11£ per 1000 units to 4700£ per 500000 units). From OpenStack we select an `m1.small` instance, with 1 GB and 1 CPU, and compute an equivalent cost of half the number of units from Flexiant (based on our assumption that OpenStack private cloud units are much cheaper than Flexiant units due to maintenance costs, e.g., electricity, or administration). The price of a network interface card, associated by default with each instance is 0.13 units, which we also set for OpenStack cloud experiment settings.

The SYBL elasticity requirements are associated with the M2M DaaS at different levels (e.g., cloud service level, service topology level). Since the cloud infrastructures are different, the requirements have to be adjusted for the providers, as they provide different performance at different costs. For Flexiant, we set a requirement of keeping response time less than 8 ms (see `Co3` for Flexiant case), while for the OpenStack private cloud we set the requirement of maintaining response time below the limit of 200 ms (see `Co3` for OpenStack case). As we have equated the costs for the two providers considering resources provided, we maintain the same cost requirement for the two cases (see `Co4`).

Figure 9 shows how the `Event Processing Service Topology` of the DaaS cloud service is affected by rSYBL control actions on the Flexiant cloud. An action enforcement is reflected in a change of cost, the deployment of a new instance with the necessary configurations being reflected in a cost increase, while the removal of an instance associated to a unit being reflected in a decrease in cost. We can see that starting from minimal deployment configuration (1 VM per service unit), rSYBL manages to find a level of resources configurations where any control enforcements do not affect the quality characteristics as it was the case in the first part of the experiment, when the response time has a short peak of 180 ms. For the second case presented in Figure 10, running on OpenStack cloud, the evolution of the service is different since the cost (i.e., the actual price which needs to be paid at the end of the day) is smaller, while the per-

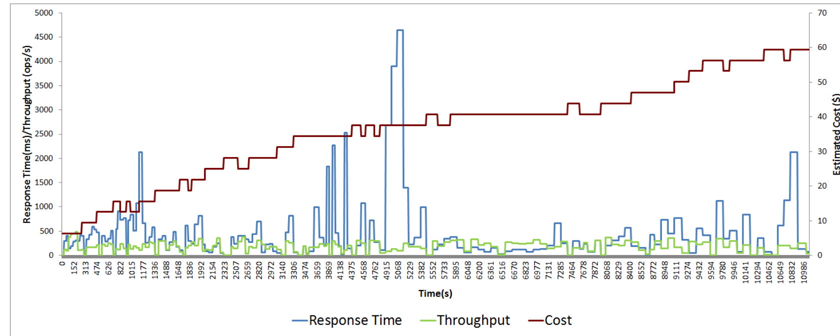

Fig. 10: Event Processing Service Topology on OpenStack-based private cloud

Config.	DB Controller	DB Nodes	Workload	Total execution time	Cost (Units)
Config1	1	3	Workload1	44 min	9.13
Config2	2	2	Workload1	28.4 min	5.88
Config1	1	3	Workload2	>3h+connection failures	$>$ 37.56
Config2	2	2	Workload2	102.75 min	21.53

Table III: Cost and execution time: comparison on different workloads

formance (e.g., response time for the `Event Processing Topology`) is as well smaller, rSYBL having to allocate a lot of resources.

6.2. Analyzing multiple levels of control: YCSB+Cassandra Cloud Service

In the second part of our experiments, we use a cloud service with two service topologies: one made from YCSB[15] Cassandra clients, the second one being a Cassandra[16] cluster, with two types of service units: Cassandra Seed (the unit acting as the controller of the cluster), and Cassandra Node. We experiment taking different level of control actions for the Cassandra NoSQL cluster. For the current experiment, the number of actions available is limited to scaling in and out at service unit level and at service topology level, by adding/removing virtual machines hosting data nodes, or by instantiating entire new data clusters, and making the proper configurations for them to receive requests from the YCSB clients.

To show the importance of higher level elasticity control, Table III presents performance and cost data on different Data Service Topology configurations and different workloads. We use two update-heavy YCSB workloads[17], the `Workload 1` having ten times less operations to be executed than `Workload 2`. We assume the OpenStack costs in experiment above (see Table II).The first important reason for enabling topology level elasticity is that multiple clusters remove the single-point of failure problem, decreasing the probability of failures, imminent for the case of highly intensive workloads with a single Cassandra seed. We show how 2 clusters (Config 2) can decrease the final cost as opposed to a single cluster (Config 1), and more importantly that it can avoid errors due to overloading. For instance, for the more intensive and longer workload, `Workload 2`, a single cassandra cluster, although with multiple virtual machines for the slave component, reaches a point where it cannot serve requests anymore, in case only the service unit level control is enabled.

[15]https://github.com/brianfrankcooper/YCSB
[16]http://cassandra.apache.org/
[17]https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads

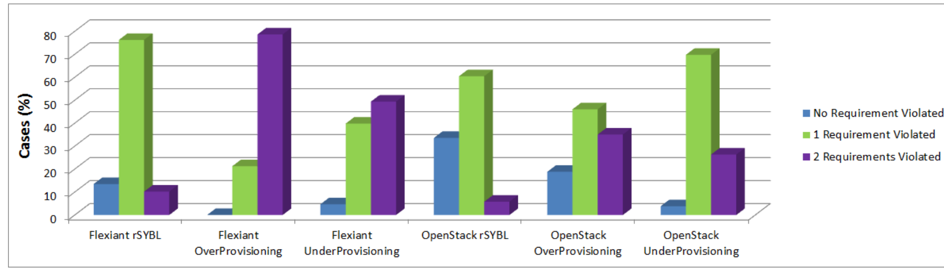

Fig. 11: Requirements fulfillment on Flexiant and OpenStack

Therefore, enabling various types of actions, and creating controllers that differentiate among them taking into consideration the effect they have not only on the current part of the cloud service which is being reconfigured, but on the overall cloud service and on various other parts as well, can greatly improve cloud services elasticity. With the capability to control service's elasticity both at service unit level and at topology level, rSYBL can improve the elasticity of cloud services both from the performance and from the cost perspective.

6.3. rSYBL performance analysis

rSYBL needs a dedicated VM, but can be collocated with other service management tools like MELA. The overhead of running rSYBL is $Cost_{VM}+Cost_{Network_eGress}$, where $Cost_{VM}$ is the cost of a VM for current cloud provider, given that we know approximately how long we would like the service hosted, and we can choose a subscription-based cost schema for this VM, and the cost of communicating among regions belonging to the same cloud provider $Cost_{Network_eGress}$, for the case we have a multi-region deployment. For the stable workload case, this cost is not justified since we have no need for elasticity, as we can create the optimum static configuration and use it. In the case the workload is variable, it makes sense to try to reduce the bigger cost which usually are connected with virtual machines and storage disks.

Controlling the service by using rSYBL empowers the user to specify the requirements s/he is interested in, at the level and for the unit which fits best, since most of the times, the user knows best what is his/her budget, and what are is the desired quality. As rSYBL's main goal is fulfilling user's requirements, we analyze the degree with which rSYBL manages to fulfill user requirements, for the M2M DaaS described in Section 6.1. We compare rSYBL control outcome with two stable cases which are manually configured for this experiment: (i) under-provisioning strategy (fixed configuration with minimum resources used with rSYBL - 4 VMs), and (ii) over-provisioning strategy (fixed configuration with maximum resources used with rSYBL - 14 VMs in Flexiant and 17 VMs in OpenStack). When computing the cost for the case of running with rSYBL control strategy, we factor in the cost for rSYBL to run as part of the M2M DaaS cost. We want to understand whether and how much the elasticity performance impact affects requirements (i.e., each control action initially decreases performance, and needs a 'cool-down period' [Gambi et al. 2013]).

Analyzing the comparison from Figure 11, we can see that rSYBL is better than both under-provisioning and over-provisioning strategies, on both cloud providers used. In the over-provisioning case, while most of the times the response time requirement is fulfilled, the one on cost (`Co4`) and the one on data end CPU usage are not fulfilled (`Co2`), due to the fact that we have a continuously high number of resources for the `Event Processing Service Topology`, for which the maximum resources of the `Data End Service Topology` allocated by rSYBL in Section 6.1 are not enough. For the current framework, rSYBL takes only reactive actions, reason for which the cases in which

one requirement is violated is quite large. We see as future work incorporating into rSYBL predictive decision making, thus decreasing the number of cases in which requirements are violated.

7. RELATED WORK

7.1. Cloud services requirements specification

Resource re-allocation and requirements specification have been a focus usually from the SLA fulfilment or scheduling and resource allocation perspectives. Hamid et al. [Fard et al. 2012] approach static scheduling with a different view, defining a multi-objective optimization algorithm and demonstrating its usefulness on real-world applications. The authors consider makespan, economic cost, energy consumption and reliability in their multi-objective list scheduling algorithm. Han et al. [Han et al. 2012] describe an approach for fine-grained scaling at resource level in addition to the VM-level scaling, which uses a lightweight scaling algorithm for improving resource utilization while reducing cloud providers' costs. Our approach differs from this research work in three main points: we support (i) multiple levels of elasticity control based on (ii) user-specific, high level requirements with (iii) multiple elasticity dimensions.

Martin et al. [Martin et al. 2011] present an attempt to tackle the problem of elasticity from the point of view of resource and elasticity in SaaS based clouds. The authors propose relating cost with quality: cost per performance metric (C/P) and cost per throughput (C/T). However, existing approaches have not developed flexible languages for controlling multi-dimensional elastic properties. [Galán et al. 2009] proposes an extension of OVF for service specification in cloud environments describing resource as well as business rules and enforcing them through resource allocation/de-allocation. Kouki et al. [Kouki et al. 2014] propose extending Cloud Service Level Agreement (CSLA) with features for cloud management, showing the degree of SLA fulfillment with and without elasticity for different cloud levels (SaaS, IaaS). In contrast with these approaches, our main focus is describing elasticity requirements and the different granularities at which they can be specified by developers, end-users or cloud providers.

The major difference between existing work and our approach from elasticity requirements specification perspective is that our work tackles elasticity requirements specification from more than one perspective (resource, quality, cost) and at different levels of granularity, thus assigning the user the capacity of specifying when the application should scale throughout its execution and, most importantly how.

7.2. Elasticity control of cloud services

Elasticity control of storage based on resources and quality has been focused by various research work, e.g., adaptively deciding how many database nodes are needed depending on the monitored data in [Tsoumakos et al. 2013]. Yu et al. [Yu and Thain 2012] propose an approach for resource management of elastic cloud workflows. They present a generic workflow architecture with components such as makeflow (that parallelizes large complex workflows on clusters grids and clouds) and master-work-workers.

Chard et al. [Chard et al. 2015] propose an approach for cost-aware heterogeneous resources provisioning for scientific workflows. The authors study the impact of using both by using spot and on-demand AWS instances, and different availability zones. Almeida et al. [Almeida et al. 2014] propose a branch and bound approach for optimally selecting services from multiple clouds during runtime. Based on the software product lines paradigm, the proposed approach scales well in selecting optimal resources, even for high number of possible configurations. In [Kranas et al. 2012] the authors propose a framework for automatic scalability using a deployment graph as a base model for the application structure. The authors introduce elasticity as a service (ElaaS), cross-

cutting different cloud stack layers (SaaS, PaaS, IaaS), to offer SLA fulfillment while decreasing operational costs.

Compared to the above-mentioned work, we control elasticity not just in terms of resources but also in terms of quality and cost and use application structure for proposing an accurate multiple level control of elasticity of cloud services. Furthermore, they lack user-customized elasticity control. We propose a user oriented elasticity control in which the user (cloud service creator, application developer, etc.) specifies how the cloud service should behave for achieving the elasticity property. Moreover, we argue in favor of an elastic services control aware of the structure of the elastic service, profiting from this knowledge for a multiple levels elasticity control of cloud services.

8. CONCLUSIONS AND FUTURE WORK

Using current state-of-the-art solutions, cloud service developers are capable to solely control virtual resources, by specifying intricate policies concerning system-level metrics. We presented a solution for multi-level cloud service elasticity control, considering requirements associated to multiple abstraction levels. rSYBL framework is open-source, extensible, and can be customized for different cloud providers and cloud services, having various preferences in terms of elasticity control. We base our control mechanisms on the user-provided requirements, and on cloud service pre-deployment information and runtime information. This way, we empower the users to steer the control by specifying their needs, at the service level they possess knowledge for (e.g., in the case s/he is aware of how or what should be controlled at the data end but not at the business end), and the level of detail which they are comfortable with (e.g., if the cloud provider knows only requirements about cost in relation to the number of clients they expect, s/he is not needed to specify response time requirements).

As future work, we will focus on integrating rSYBL framework with techniques for accurately estimating effects, and employing these estimations for better control of cloud services elasticity. These improvements will also open the road for predictive control, and for studying the conditions under which the predictive or reactive control is better.

REFERENCES

A Almeida, F. Dantas, E. Cavalcante, and T. Batista. 2014. A Branch-and-Bound Algorithm for Autonomic Adaptation of Multi-cloud Applications. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 315–323. DOI:http://dx.doi.org/10.1109/CCGrid.2014.25

V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. 2013. How to Adapt Applications for the Cloud Environment. *Computing* 95 (2013), 493–535. DOI:http://dx.doi.org/10.1007/s00607-012-0248-2

R. Chard, K. Chard, K. Bubendorfer, L. Lacinski, R. Madduri, and I. Foster. 2015. Cost-Aware Elastic Cloud Provisioning for Scientific Workloads. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. 971–974. DOI:http://dx.doi.org/10.1109/CLOUD.2015.130

G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar. 2013a. Multi-level Elasticity Control of Cloud Services. In *Service-Oriented Computing*, Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu (Eds.). Lecture Notes in Computer Science, Vol. 8274. Springer Berlin Heidelberg, 429–436. DOI:http://dx.doi.org/10.1007/978-3-642-45005-1_31

G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar. 2013b. SYBL: an Extensible Language for Controlling Elasticity in Cloud Applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, 112–119.

S. Dustdar, Y. Guo, B. Satzger, and Hong-Linh Truong. 2011. Principles of Elastic Processes. *IEEE Internet Computing* 15, 5 (sept.-oct. 2011), 66 –71. DOI:http://dx.doi.org/10.1109/MIC.2011.121

H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer. 2012. A Multi-objective Approach for Workflow Scheduling in Heterogeneous Environments. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) (CCGRID '12)*. IEEE Computer Society, Washington, DC, USA, 300–309. DOI:http://dx.doi.org/10.1109/CCGrid.2012.114

F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, and L. M. Vaquero. 2009. Service specification in cloud environments based on extensions to open standards. In *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE (COMSWARE '09)*. ACM, New York, NY, USA, Article 19, 12 pages. DOI:http://dx.doi.org/10.1145/1621890.1621915

A Gambi, D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar. 2013. On estimating actuation delays in elastic computing systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*. 33–42. DOI:http://dx.doi.org/10.1109/SEAMS.2013.6595490

R. Han, L. Guo, M. M. Ghanem, and Y. Guo. 2012. Lightweight Resource Scaling for Cloud Applications. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012) (CCGRID '12)*. IEEE Computer Society, Washington, DC, USA, 644–651. DOI:http://dx.doi.org/10.1109/CCGrid.2012.52

C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, and S. Dustdar. 2014. MADCAT - A Methodology for Architecture and Deployment of Cloud Application Topologies. In *8th International Symposium on Service-Oriented System Engineering*. IEEE.

Y. Kouki, F.A De Oliveira, S. Dupont, and T. Ledoux. 2014. A Language Support for Cloud Elasticity Management. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 206–215. DOI:http://dx.doi.org/10.1109/CCGrid.2014.17

P. Kranas, V. Anagnostopoulos, A. Menychtas, and T. Varvarigou. 2012. ElaaS: An Innovative Elasticity as a Service Framework for Dynamic Management across the Cloud Stack Layers. In *2012 Sixth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*. 1042 –1049. DOI:http://dx.doi.org/10.1109/CISIS.2012.117

P. Martin, A. Brown, W. Powley, and J. L. Vazquez-Poletti. 2011. Autonomic management of elastic services in the cloud. In *Proceedings of the 2011 IEEE Symposium on Computers and Communications (ISCC '11)*. IEEE Computer Society, Washington, DC, USA, 135–140. DOI:http://dx.doi.org/10.1109/ISCC.2011.5984006

D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar. 2013. MELA: Monitoring and Analyzing Elasticity of Cloud Services. In *2013 IEEE Fifth International Conference on Cloud Computing Technology and Science (CloudCom)*. 80–87. DOI:http://dx.doi.org/10.1109/CloudCom.2013.18

S. Tai, P. Leitner, and S. Dustdar. 2012. Design by Units: Abstractions for Human and Compute Resources for Elastic Systems. *IEEE Internet Computing* 16, 4 (2012), 84–88. DOI:http://dx.doi.org/10.1109/MIC.2012.81

D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. 2013. Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, 34–41.

L. Yu and D. Thain. 2012. Resource Management for Elastic Cloud Workflows. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 775–780. DOI:http://dx.doi.org/10.1109/CCGrid.2012.107